Introduction to Python

Course Notes

Programming Explained in 5 Minutes

The computer understands 1s and 0s only. To communicate a real-life problem to the computer, you need to create a specific type of text, called a **source code** or a **human readable code**, that software can read and then process to the computer in 1s and 0s.

Term	Definition
program	a sequence of instructions that designate how to execute a computation
programming	taking a task and writing it down in a programming language that the computer can understand and execute

Why Python?

Python is an *open-source, general-purpose high-level* programming language.

Term	Definition
Open-source software (OSS)	Open-source means it is free. Python has a large and active scientific community with access to the software's source code and contributes to its continuous development and upgrading, depending on users' needs.
General-purpose	There is a broad set of fields where Python could be applied – web programming, analysis of financial data, analysis of big data, and more.
High-level	High-level languages employ syntax a lot closer to human logic, which makes the language easier to learn and implement.

Python's popularity lies on two main pillars. One is that it is an easy-to-learn programming language designed to be highly readable, with a syntax quite clear and intuitive. And the second reason is its user-friendliness does not take away from its strength. Python can execute a variety of complex computations and is one of the most powerful programming languages preferred by specialists.

Why Jupyter?

The **Jupyter Notebook App** is a server-client application that allows you to edit your code through a web browser.

Language kernels are programs designed to read and execute code in a specific programming language, like Python, R, or Julia. The Jupyter installation always comes with an installed Python kernel, and the other kernels can be installed additionally.

The **Interfaces**, where you can write code, represent the clients. An example of such a client is the *web browser*.

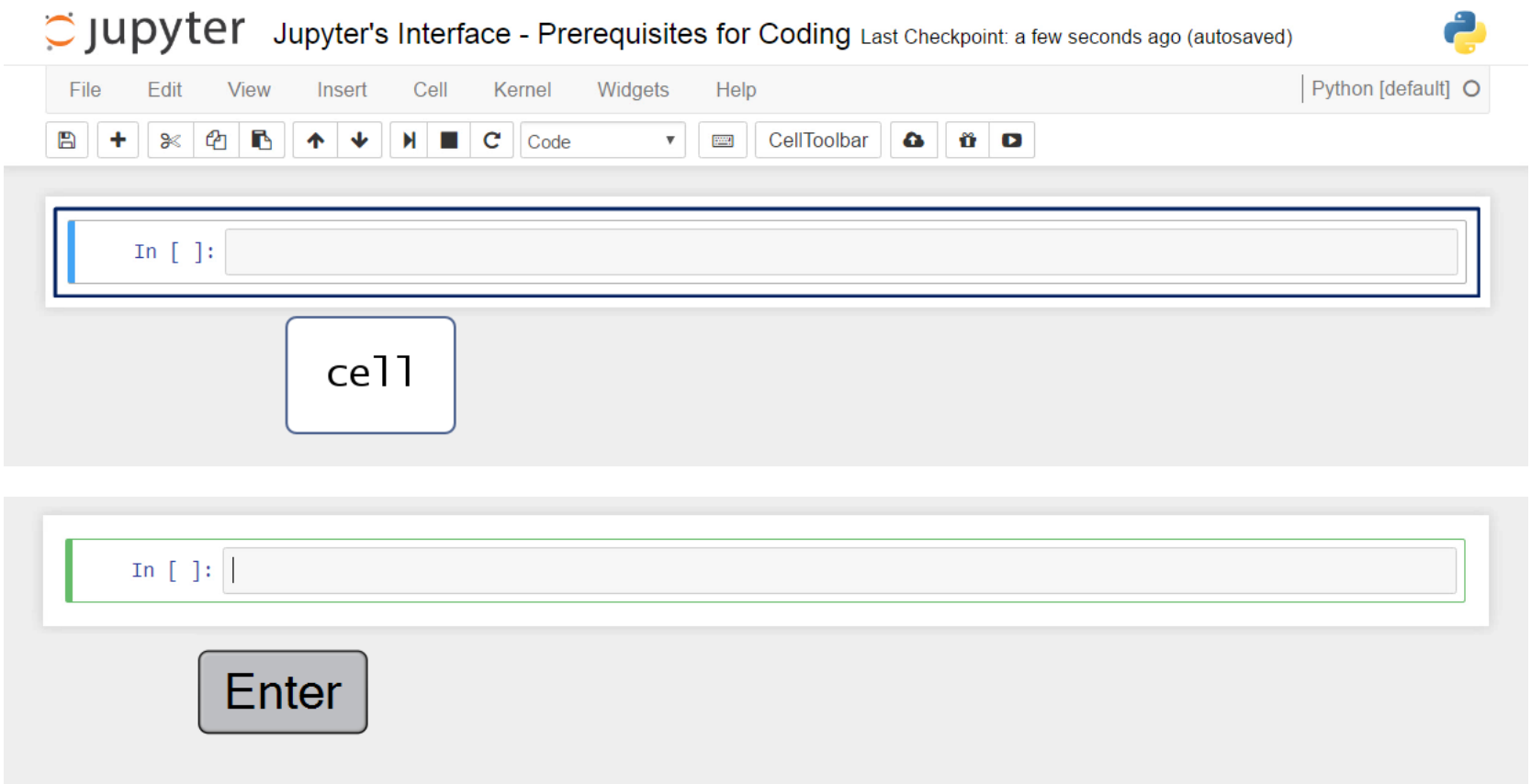
The Jupyter server provides the environment where a *client* is matched with a corresponding *languages kernel*. In our case, we will focus on *Python*, and a *web browser* as a client.

Jupyter's Interface – the Dashboard

As soon as you load the notebook, the **Jupyter dashboard** opens. Each file and directory has a check box next to it. By ticking and unticking an item, you could manipulate the respective object – that means you can duplicate or shutdown a running file.

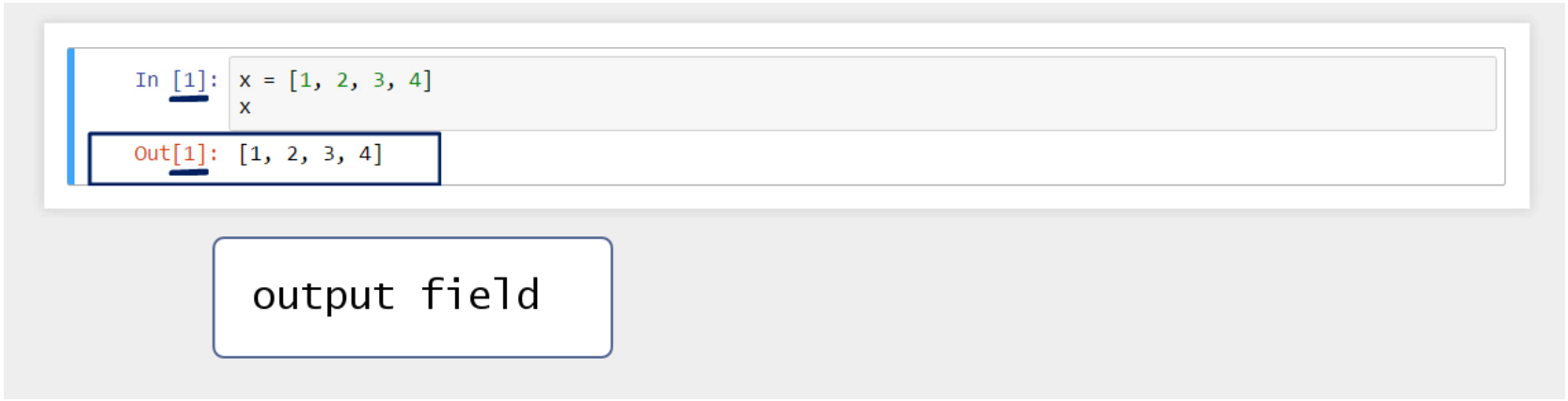
From the *Upload* button in the top-right corner, you can upload a notebook into the directory you are in.
You can expand the *New* button. From the list that falls, you will most likely need to create a new text file, a new folder, or a new notebook file

Jupyter's Interface – Prerequisites for Coding



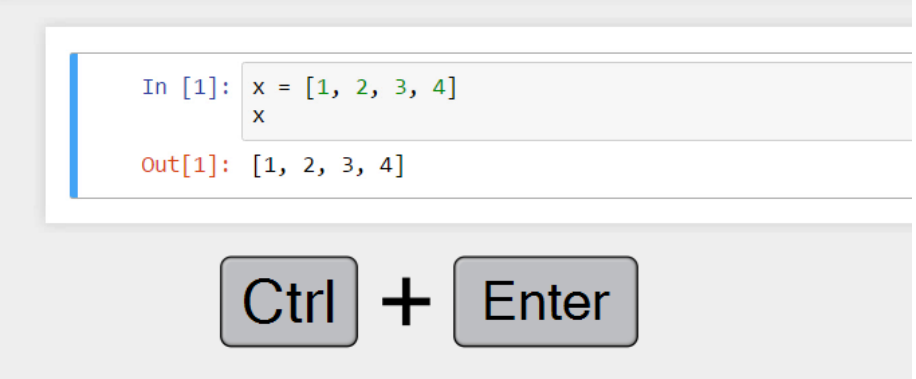
You can access a cell by pressing "Enter". Once you've done that, you'll be able to see the cursor, so you can start typing code.

Jupyter's Interface – Prerequisites for Coding



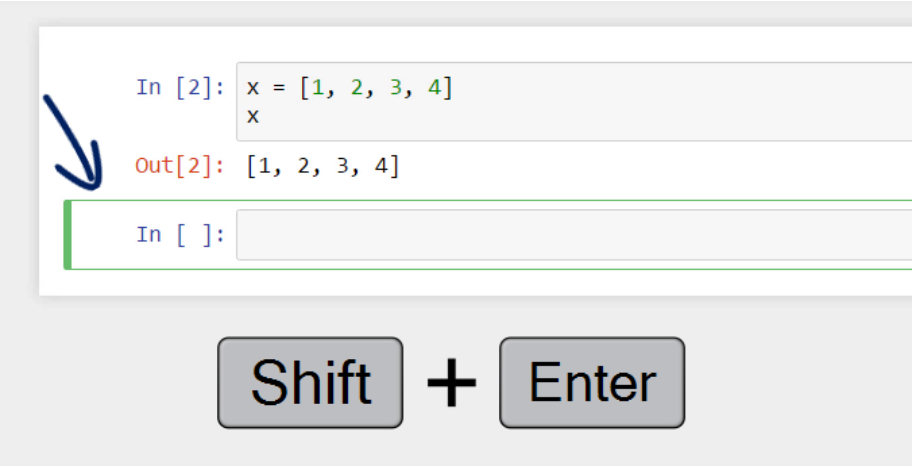
365 DataScience

Jupyter's Interface – Prerequisites for Coding



You can execute a command in two ways.

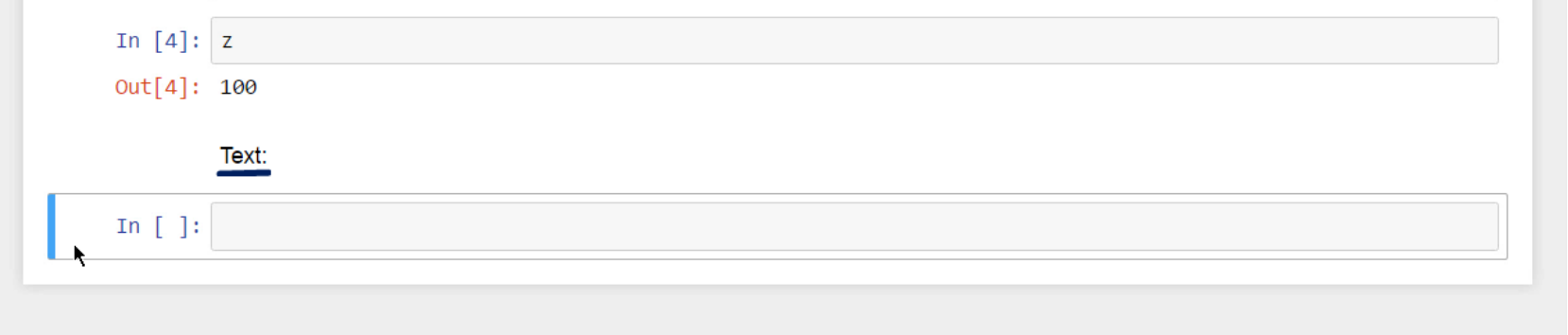
The first one is to hold Ctrl and then press Enter. By doing this, the machine will execute the code in the cell, and you will "stay" there, meaning I will not have created or selected another cell.



The second option allows for a more fluid code writing. To execute the same code, hold "Shift" and then press "Enter". The previous two commands are being executed and then a new cell where you can write code is created.

If you use "Shift" and "Enter", you can continue typing code easily.

Jupyter's Interface – Prerequisites for Coding



A **markdown cell** is a cell that contains strictly documentation - text not executed as a code. It will contain some message you would like to leave to the reader of the file.

Variables

One of the main concepts in programming is **variables**. They are your best friends. You will deal with them all the time. You will use them to store information. They will represent your data input.

```
In [1]: x = 5

In [2]: x
Out[2]: 5
```

Let's say you want to have a variable x that is equal to the value of 5 and then ask the computer to tell you the value of that variable. Type x equals 5.
Write x and then execute. And here's the result – 5.

```
In [3]: y = 8

In [5]: print y
        8
```

An alternative way to execute the instruction that will provide the value we assigned to y would be to use the *print* command.
If we say "print y", the machine will simply execute this command and provide the value of y *as a statement*.

Numbers and Boolean Values

When programming, not only in Python, if you say that a variable has a numeric value, you are being ambiguous. The reason is that numbers can be **integers** or floating points, also called **floats**, for instance.

Term	Definition
Integer	Positive or negative whole numbers without a decimal point *Example: 5, 10, -3, -15*
Floating point (float)	Real numbers. Hence, they have a decimal point *Example: 4.75, -5.50, 11.0*
Boolean value	a True or False value, corresponding to the machine's logic of understanding 1s and 0s, on or off, right or wrong, true or false. *Example: True, False*

Strings

Strings are text values composed of a sequence of characters.

```
In [2]: 'George'
Out[2]: 'George'

In [3]: "George"
Out[3]: 'George'

In [4]: print 'George'
George

In [5]: print "George"
George
```

Type single or double quotation marks around the name George. Python displays ‘George’ if you don’t use the print command. Should you use print, the output will be shown with no quotes – you’ll be able to see plain text.

Strings

Strings are text values composed of a sequence of characters.

```
In [11]: "I'm fine"
Out[11]: "I'm fine"

In [12]: 'I\'m fine'
Out[12]: "I'm fine"

In [13]: 'Press "Enter"'
Out[13]: 'Press "Enter"'
```

To type “I’m fine” you’ll need the apostrophe in the English syntax, not for the Pythonic one.
In such situations, you can distinguish between the two symbols – put the text within double quotes and leave the apostrophe, which technically coincides with the single quote between I and M.

An alternative way to do that would be to leave the quotes on the sides and place a back slash before the apostrophe within the phrase. This backslash is called an **escape character**, as it changes the interpretation of characters immediately after it.

To state “press “Enter””, the outer symbols must differ from the inner ones. Put single quotes on the sides.

Strings

Strings are text values composed of a sequence of characters.

```
In [17]: print 'Red ' + 'car'

         Red car

In [18]: print 'Red', 'car'

         Red car
```

Say you wish to print “Red car” on the same line. “Add” one of the strings to the other by typing in a plus sign between the two. Put a blank space before the second apostrophe of the first word. (In [17])

Type “print ‘Red’", and then put a comma, which is called a **trailing comma**, and Python will print the next word, ‘car’, on the same line, separating the two words with a blank space. (In [18])

Arithmetic Operators

```
In [1]: 1 + 2
Out[1]: 3
```

In the equation you see here, 1 and 2 are called **operands**, The plus and minus signs are called **operators**, and given they also represent arithmetic operations, they can be called **arithmetic operators**.

Operator	Description
+	Addition
-	Subtraction
/	Division *Note: If you want to divide 16 by 3, when you use Python 2, you should look for the quotient of the float 16 divided by 3 and not of the integer 16 divided by 3. So, you should either transform the number into a float or type it as a float directly.*
%	Returns remainder
*	Multiplication
**	Performs power calculation

The Double Equality Sign

```
In [1]: y = 5 ** 3

In [2]: y
Out[2]: 125

In [3]: y == 125
Out[3]: True

In [4]: y == 126
Out[4]: False
```

The *equals* sign when programming means "assign" or "bind to". For instance, "assign 5 to the power of 3 to the variable y"; "bind 5 to the power of 3 to y". From that moment for the computer, y will be equal to 125.

When you mean equality between values and not assignment of values in Python, you'll need **the double equality sign**. Anytime you use it, you will obtain one of the two possible outcomes – "True" or "False".

Reassign Values

```
In [1]: z = 1
        z

Out[1]: 1

In [2]: z = 3
        z

Out[2]: 3

In [3]: z + 5

Out[3]: 8

In [4]: z = 7
        z

Out[4]: 7
```

If you assign the value of 1 to a variable z, your output after executing z will be 1. After that, if you assign 3 to the same variable z, z will be equal to 3, not 1 anymore.

Python **reassigns** values to its objects. Therefore, remember the last command is valid, and older commands are overwritten.

Add Comments

```
In [1]: #This is just a comment and not code!
        print 7,2

        7 2

In [2]: #Comment 1
        #Comment 2
        print 1

        1
```

Comments are sentences not executed by the computer; it doesn't read them as instructions. The trick is to put a *hash sign* at the beginning of each line you would like to insert as a comment.

If you would like to leave a comment on two lines, don't forget to place the hash sign at the beginning of each line.

Line Continuation

```
In [ ]: 2.0 * 1.5 + \
        5
```

You might prefer to send part of the code to the next line. So, 2.0 times 1.5 plus 5 could be written in two lines, and the machine could still read it as one command. This could be achieved by putting a back slash where you would like the end of the first line to be. It indicates you will continue the same command on a new line.

Indexing Elements

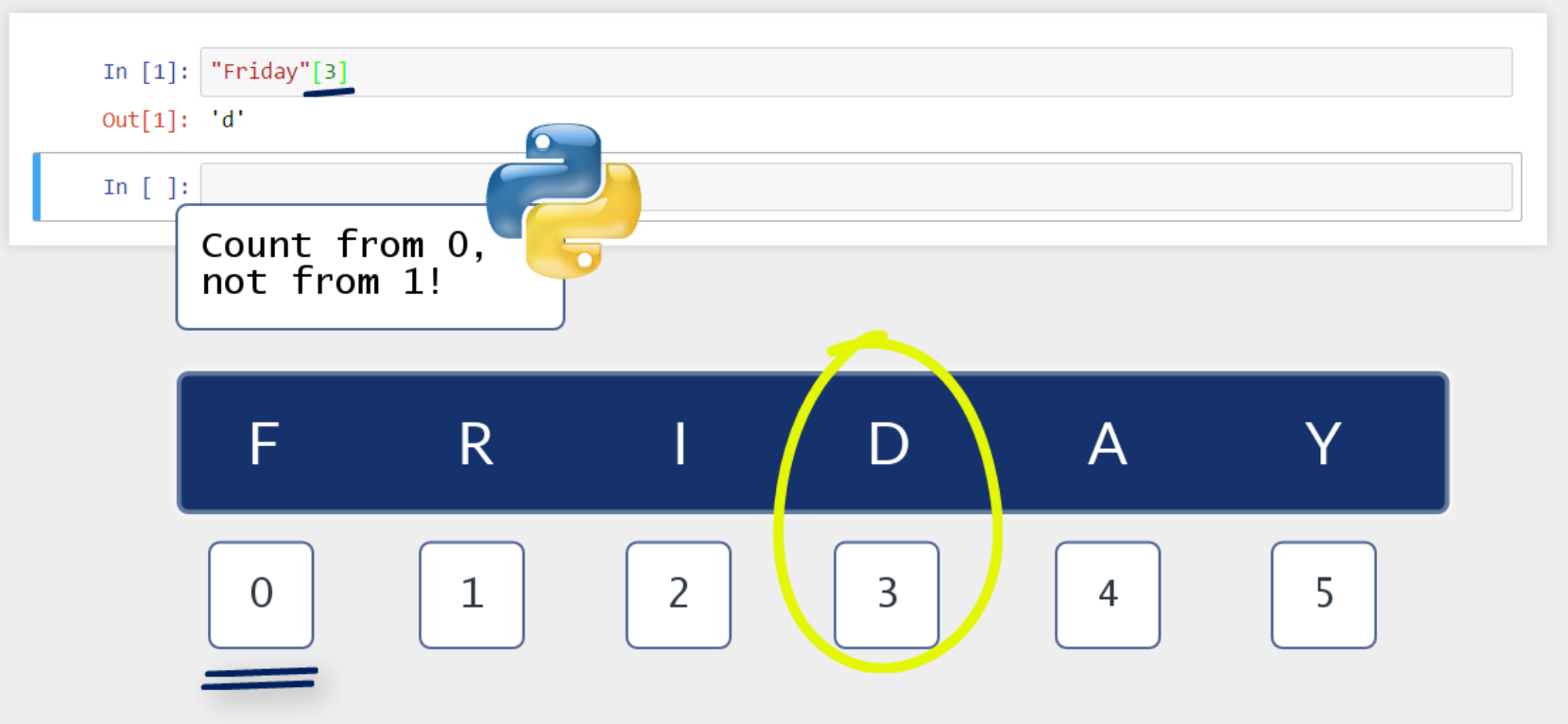
```
In [ ]: "Friday"[]
```

```
"Name_of_variable"[index_of_element]
```

Is it possible to extract the letter “d”?
Yes, you can do that *by using square brackets*. And within them, you should specify the position of the letter we would like to be extracted.

Note:
Make sure you don’t mistake brackets for parentheses or braces:
parentheses – ()
brackets – []
braces – {} !

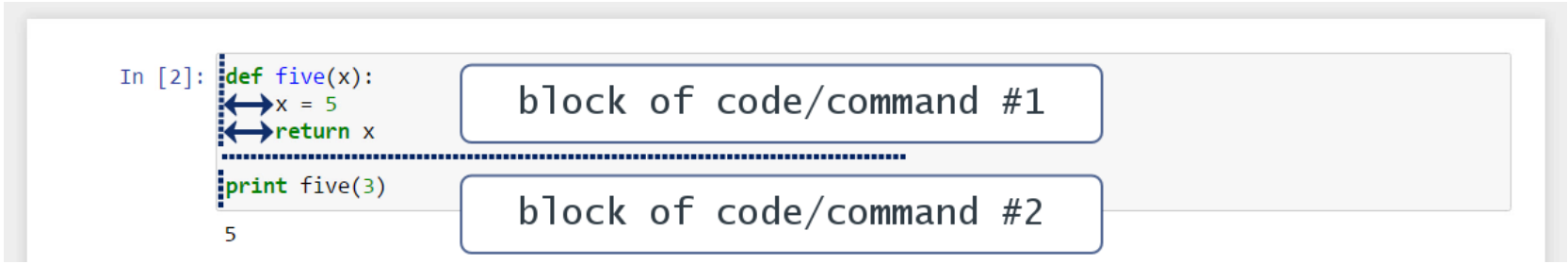
Indexing Elements



A very important thing you should remember is that, **in Python, we count from 0, not from 1!** 0, 1, 2, 3, 4, and so on. That's why I'll ask for the 4th letter, 'd', by writing 3 here.

Structure Your Code with Indentation

The way you apply **indentation** in practice is important, as this will be the only way to communicate your ideas to the machine properly.



Def and *Print* form two separate and, written in this way, clearly distinguishable **blocks of code** or **blocks of commands**.

Everything that regards the function is written with one indentation to the inside. Once you decide to code something else, start on a new line with no indentation. The blocks of code are more visible, and this clarifies the logic you are applying to solve your problem.

Comparison Operators

Operator	Description
==	Verifies the left and right side of an equality are equal
!=	Verifies the left and right side of an equality are not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Logical Operators

Briefly, the **logical operators** in Python are the words "**not**", "**and**", and "**or**". They compare a certain amount of statements and return Boolean values – "True" or "False" – hence their second name, **Boolean operators**.

Operator	Description
"And"	Checks whether the two statements around it are "True" *Example: "True and False" leads to True*
"Or"	Checks whether at least one of the two statements is "True" *Example: "False or True" leads to True*
"Not"	Leads to the opposite of the given statement *Example: "not True" leads to False*

You must respect the **order of importance** of these three operators. It is: "not" comes first, then we have "and", and finally "or".

Identity Operators

The **identity operators** are the words "**is**" and "**is not**". They function similar to the double equality sign and the exclamation mark and equality sign we saw earlier.

```
In [1]: 5 is 6
Out[1]: False

In [2]: 5 == 6
Out[2]: False

In [3]: 5 is not 6
Out[3]: True

In [4]: 5 != 6
Out[4]: True
```

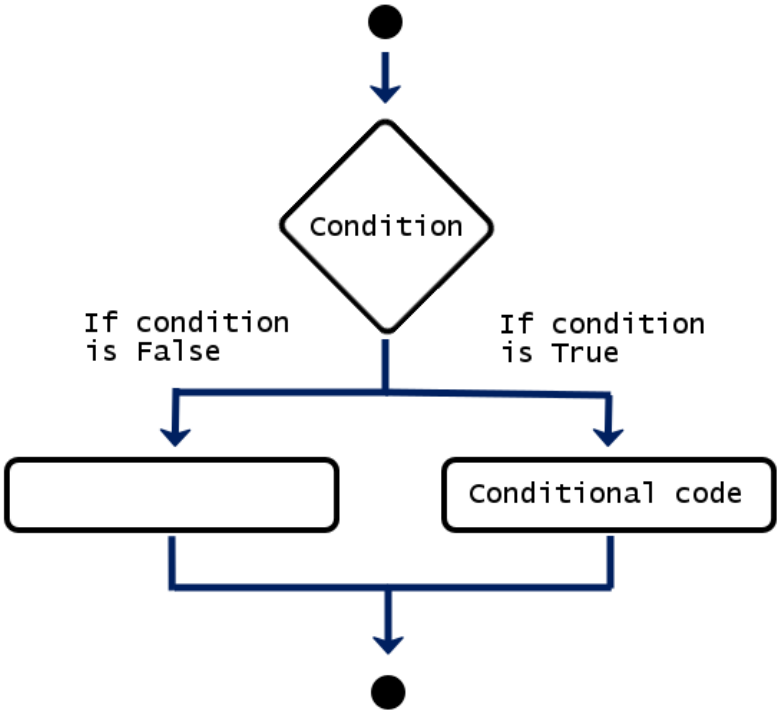
Introduction to the IF statement

A prominent example of **conditional statements** in Python is the **“If” statement**. What you can use it for is intuitive, but it is very important to learn the syntax.

```
In [ ]: if 5 == 15 / 3:
            print "Hooray!"
```

```
if condition:
    conditional code
```

Introduction to the IF statement



The graph could help you imagine the process of the conditionals. Before it displays the outcome of the operation, the machine follows these logical steps. If the conditional code is not to be executed because the if-condition is not true, the program will directly lead you to some other output or, as it is in this case, to nothing. After any of the two situations, the machine will go to the next black point and will progress from there on.

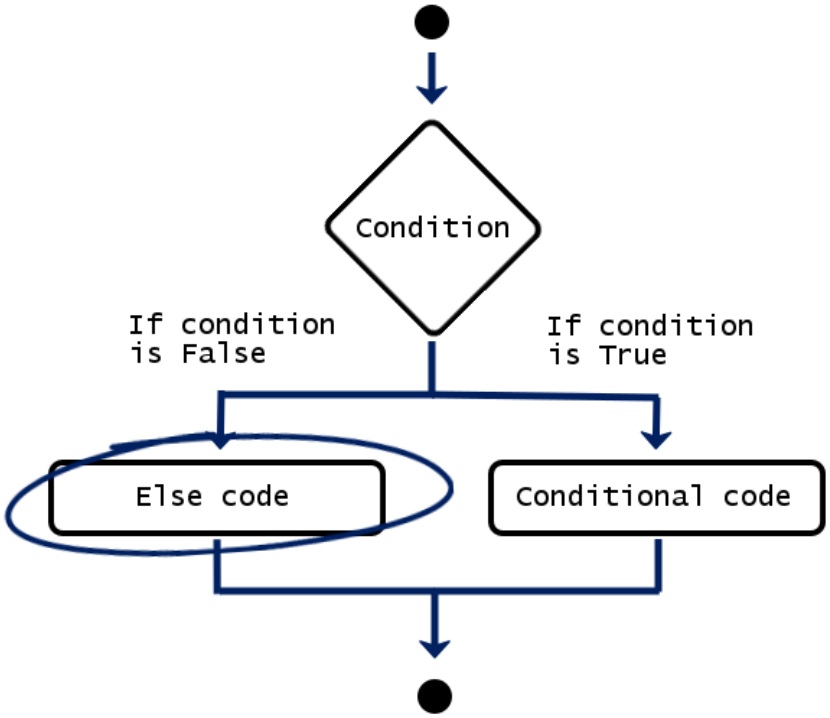
Add an ELSE statement

"**Else**" will tell the computer to execute the successive command in all other cases.

```
In [1]: x = 1

        if x > 3:
            print "Case 1"
        else:
            print "Case 2"

        Case 2
```

```
if condition:
    conditional code
else:
    else code
```

Add an ELSE statement



Instead of leading to no output, if the condition is false, we will get to an *else code*. Regardless whether the initial condition is satisfied, we will get to the end point, so the computer has concluded the entire operation and is ready to execute a new one.

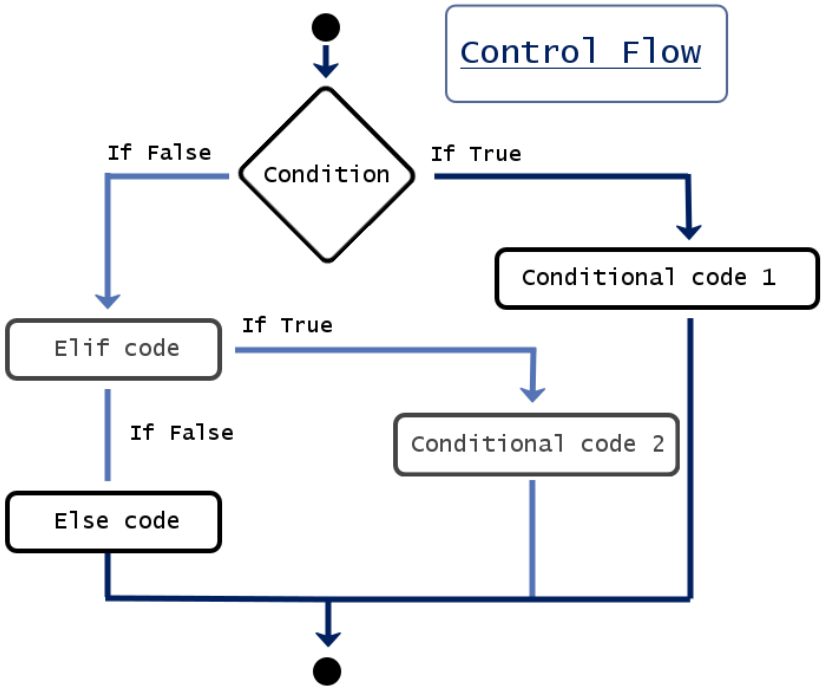
Else if, for Brief - ELIF

If y is not greater than 5, the computer will think: “else if y is less than 5”, written “**elif** y is less than 5”, then I will print out “Less”.

```
In [1]: def compare_to_five(y):
            if y > 5:
                return "Greater"
            elif y < 5:
                return "Less"
            else:
                return "Equal"
```

Know that you can add as many elif statements as you need.

Else if, for Brief - ELIF



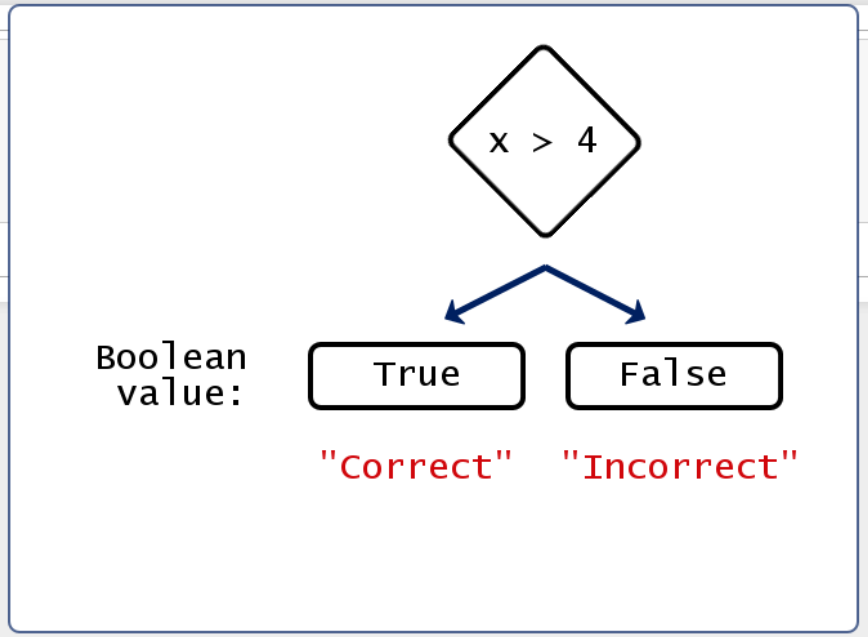
A very important detail you should try to remember is the computer always reads your commands from top to bottom. Regardless of the speed at which it works, it executes only one command at a time. Scientifically speaking, the instructions we give to the machine are part of a control flow. This is something like the flow of the logical thought of the computer, the way the computer thinks – step by step, executing the steps in a rigid order.

When it works with a conditional statement, the computer's task will be to execute a specific command once a certain condition has been satisfied. It will read your commands from the if-statement at the top, through the elif-statements in the middle, to the else- statement at the end. The first moment the machine finds a satisfied condition, it will print the respective output and will execute no other part of the code from this conditional.

A Note on Boolean Values

From a certain perspective, everything in a computer system is Boolean, comprising sequences of 0s and 1s, “False” and “True”. This is why we are paying attention to the Boolean value. It helps us understand general computational logic and the way conditionals work in Python.

```
In [1]: x = 2

        if x > 4:
            print "Correct"
        else:
            print "Incorrect"

        Incorrect
```



Basically, after you insert your if-statement, the computer will attach a Boolean value to it. Depending on the value of its outcome, “True” or “False”, it will produce one of the suggested outputs, “Correct” or “Incorrect”.

Defining a Function in Python

Python's **functions** are an invaluable tool for programmers.

```
In [1]: def simple():
            print "My first function"
```

```
def  function_name  (parameters):
⟷  function body
```

To tell the computer you are about to create a function, just write **def** at the beginning of the line. Def is neither a command nor a function. It is a **keyword**. To indicate this, Jupyter will automatically change its font color to green. Then, you can type the **name of the function** you will use. Then you can add a pair of **parentheses**. Technically, within these parentheses, you could place the **parameters** of the function if it requires you to have any. It is no problem to have a function with zero parameters.

To proceed, don't miss to put a **colon** after the name of the function.

Since it is inconvenient to continue on the same line when the function becomes longer, it is much better to build the habit of laying the instructions on a new line, with an **indent** again. Good legibility counts for a good style of coding!

Creating a Function with a Parameter

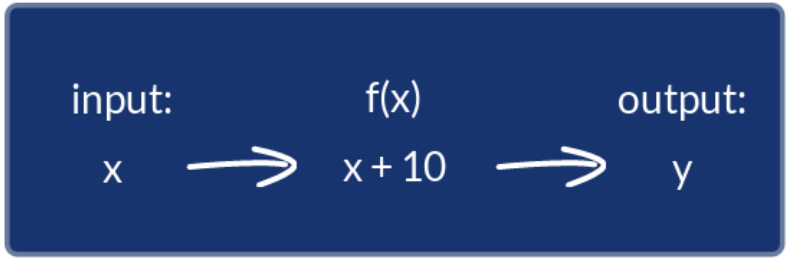
```
In [3]: def plus_ten(a):
            return a + 10

In [4]: plus_ten(2)
Out[4]: 12

In [5]: plus_ten(5)
Out[5]: 15

In [ ]:
```

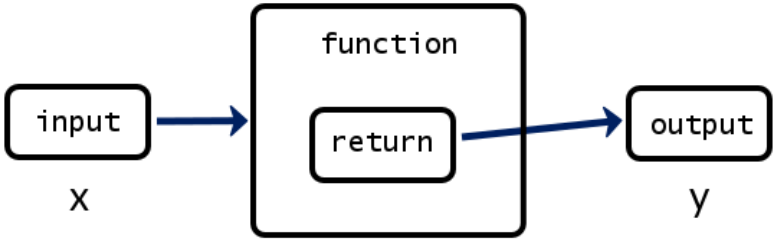
```
def  function_name  (parameters):

but call

plus_ten (arguments):
```

Don't forget to ***return*** a value from the function. We will need *plus_ten(a)* to do a specific calculation for us and not just print something.

Pay attention to the following. When we define a function, we specify in parentheses a **parameter**. In the *plus_ten()* function, "a" is a parameter. Later, when we call this function, it is correct to say we provide an **argument**, not a parameter. So we can say "call *plus_ten()* with an argument of 2, call *plus_ten()* with an argument of 5".

Creating a Function with a Parameter

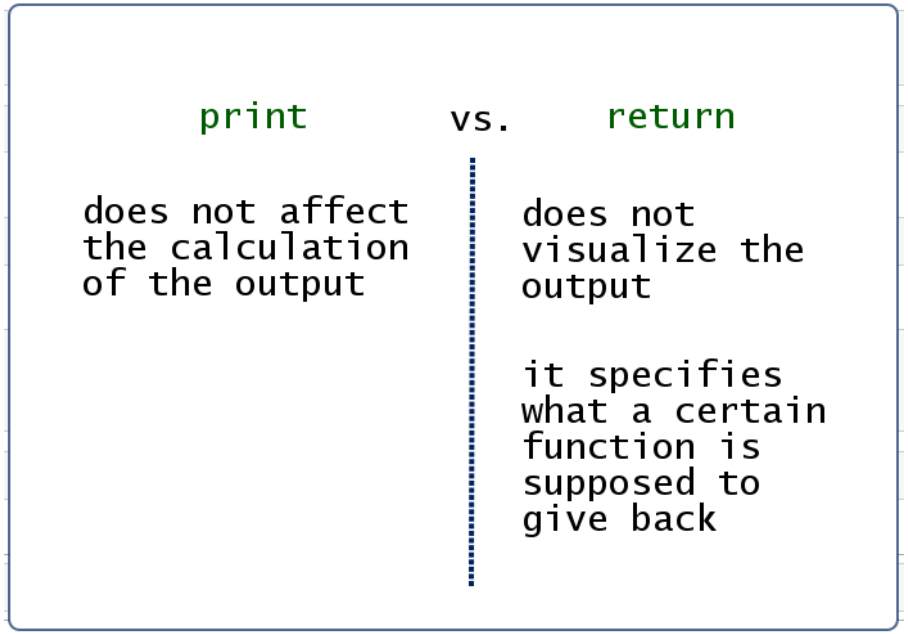




In programming, **return** regards the value of y; it just says to the machine "after the operations executed by the function f, return to me the value of y". "Return" plays a connection between the second and the third step of the process. In other words, *a function can take an input of one or more variables and return a **single** output composed of one or more values*. This is why "return" can be used only once in a function.

People often confuse print and return, and the type of situations when we can apply them.

print vs. return


print
vs.
return
does not affect the calculation of the output
does not visualize the output
it specifies what a certain function is supposed to give back

"Print" takes a statement or, better, an object, and provides its printed representation in the output cell. It just makes a certain statement visible to the programmer. Otherwise, print does not affect the calculation of the output.

Differently, return does not visualize the output. It specifies what a certain function is supposed to give back. It's important you understand what each of the two keywords does. This will help you a great deal when working with functions.

Using a Function in Another Function

```
In [ ]: def wage(w_hours):
            return w_hours * 25

        def with_bonus(w_hours):
            return wage(w_hours) + 50
```

It isn't a secret we can have a function within the function.

In *with_bonus(w_hours),* you can return directly the wage with working hours as an output, which would be the value obtained after the wage function has been run, plus 50.

Creating Functions Containing a Few Arguments

You can work with more than one parameter in a function. The way this is done in Python is by enlisting all the arguments within the parentheses, separated by a comma.

```
In [1]: def subtract_bc(a,b,c):
            result = a - b*c
            print 'Parameter a equals', a
            print 'Parameter b equals', b
            print 'Parameter c equals', c
            return result
```

```
def  function_name (parameter #1, parameter #2, ... ):
```

Creating Functions Containing a Few Arguments

```
In [2]: subtract_bc(10,3,2)

        Parameter a equals 10
        Parameter b equals 3
        Parameter c equals 2

Out[2]: 4
```

```
In [ ]:
```

You can call the function for, say, 10, 3, and 2. You will get 4.
Just be careful with the *order* in which you state the values. In this case, we assigned 10 to the variable a, 3 to b, and 2 to c.

```
In [3]: subtract_bc(b=3,a=10,c=2)

        Parameter a equals 10
        Parameter b equals 3
        Parameter c equals 2

Out[3]: 4
```

Otherwise, the order won't matter if and only if you specify the names of the variables within the parentheses.

Notable Built-In Functions in Python

When you install Python on your computer, you are also installing some of its **built-in functions**. This means you won't need to type their code every time you use them – these functions are already on your computer and can be applied directly.

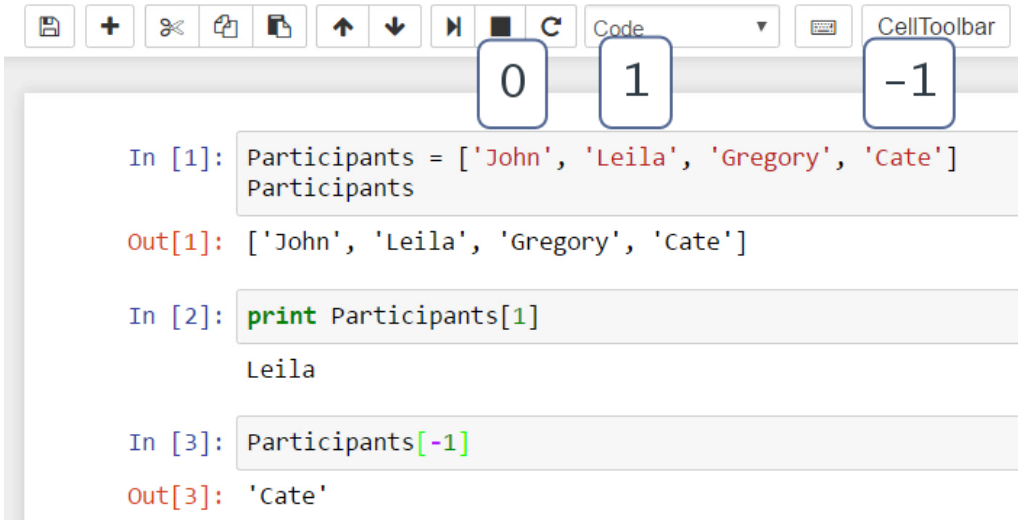
Function	Description
type()	obtains the type of variable you use as an argument
int()	transforms its argument in an *integer* data type
float()	transforms its argument in a *float* data type
str()	transforms its argument in a *string* data type
max()	Returns the highest value from a sequence of numbers
min()	Returns the lowest value from a sequence of numbers
abs()	Allows you to obtain the absolute value of its argument
sum()	Calculates the sum of all the elements in a list designated as an argument

Notable Built-In Functions in Python

Function	Description
round(x,y)	returns the float of its argument (x), rounded to a specified number of digits (y) after the decimal point
pow(x,y)	returns x to the power of y
len()	returns the number of elements in an object

Lists

A **list** is a type of sequence of data points such as floats, integers, or strings.



You can access the *Participants* list by indexing the value 1. This means you have extracted the second of the elements in this list variable ['Leila'].

In addition, there is a way to get to the last element from your list – start counting from the end towards the beginning. Then, you'd need the minus sign before the digit and don't fall in the trap of thinking we begin enumerating from 0 again! To obtain "Cate", you have to write -1.

Help Yourself with Methods

Here is the syntax that allows you to call ready-made **built-in methods** that you do not have to create on your own and can be used in Python directly.

After the name of the **object**, which in this case is the "Participants" list, you must put a dot called a **dot operator**. The dot operator allows you to **call** on or **invoke** a certain method. To call the method "append", state its name, followed by **parentheses**.

```
object . method ()
```

```
In [8]: Participants.append("Dwayne")
        Participants

Out[8]: ['John', 'Leila', 'Maria', 'Dwayne']
```

To insert the name "Dwayne" in our list, you must put the string "Dwayne" in inverted commas between the parentheses.

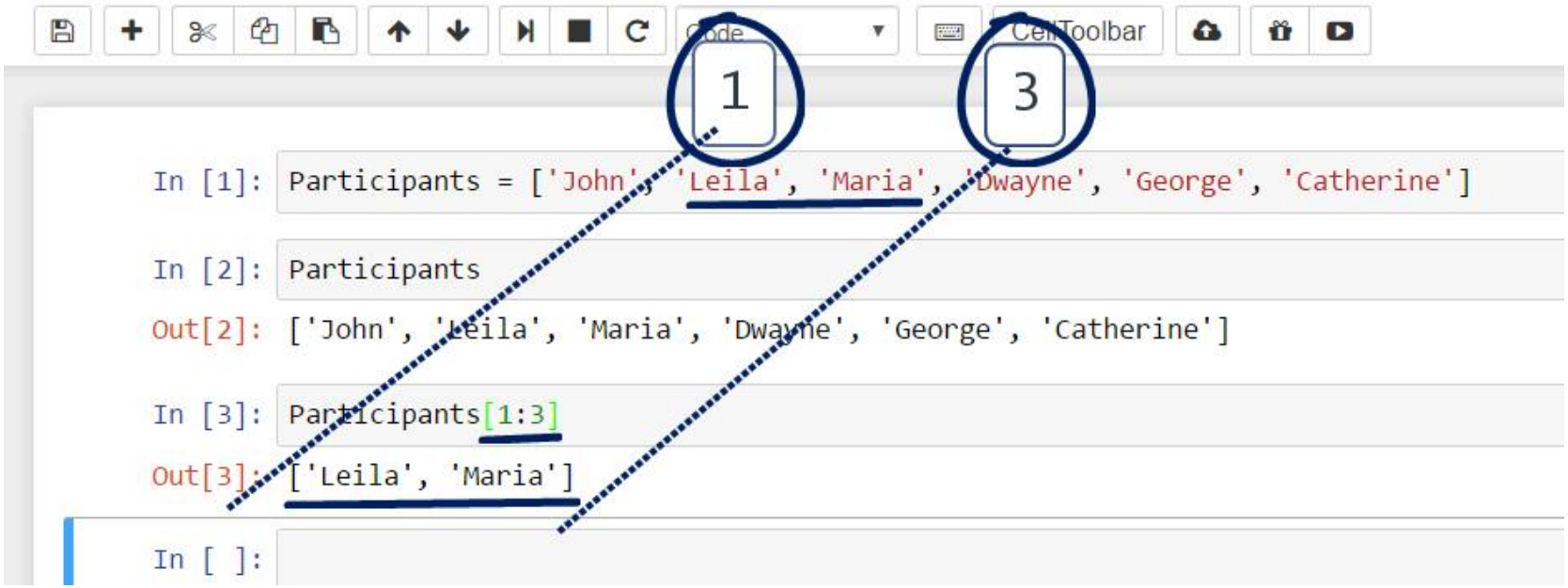
Help Yourself with Methods

```
In [9]: Participants.extend(['George', 'Catherine'])
        Participants

Out[9]: ['John', 'Leila', 'Maria', 'Dwayne', 'George', 'Catherine']
```

Alternatively, the same result can be achieved by using the “extend” method. This time, within the parentheses, you’ll have to add brackets, as you are going to extend the “Participants” list by adding a list specified precisely in these parentheses.

List Slicing

Many of the problems that must be solved will regard a tiny portion of the data, and in such cases, you can apply slicing.



Imagine you want to use the “Participants” list to obtain a second much smaller list that contains only two names - Leila and Maria. In Pythonic, that would mean to extract the elements from the first and second position. To access these elements, we will open square brackets, just as we did with indexing, and write 1 colon 3. The first number corresponds precisely to the first position of interest, while the second number is one position above the last position we need.

Tuples

Tuples are another type of data sequences, but differently to lists, they are ***immutable***. Tuples cannot be changed or modified; you cannot append or delete elements.

```
In [1]: x = (40,41,42)
        x

Out[1]: (40, 41, 42)

In [2]: y = 50,51,52
        y

Out[2]: (50, 51, 52)

In [3]: a,b,c = 1,4,6
        c

Out[3]: 6
```

The syntax that indicates you are having a tuple and not a list is that the tuple's elements are placed within parentheses and not brackets.

The tuple is the default sequence type in Python, so if you enlist three values here, the computer will perceive the new variable as a tuple. We could also say the three values will be **packed** into a tuple.

For the same reason, you can assign a number of values to the same number of variables. On the left side of the equality sign, add a tuple of variables, and on the right, a tuple of values. That's why the relevant technical term for this activity is **tuple assignment**.

Dictionaries

Dictionaries represent another way of storing data.

```
In [1]: dict = {'k1': "cat", 'k2': "dog", 'k3': "mouse", 'k4': "fish"}
        dict
Out[1]: {'k1': 'cat', 'k2': 'dog', 'k3': 'mouse', 'k4': 'fish'}

In [2]: dict['k1']
Out[2]: 'cat'
```

Each value is associated with a certain **key**. More precisely, a key and its respective value form a ***key-value pair***. After a certain dictionary has been created, a value can be accessed by its key, instead of its index!

```
In [ ]: dict['k5'] = 'parrot'
```

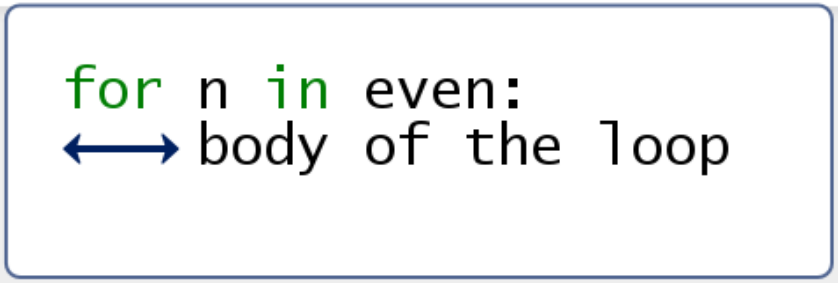
```
dictionary_name [new_key_name] = new_value_name
```

Similarly, as we could do with lists, we can add a new value to the dictionary in the following way: the structure to apply here is dictionary name, new key name within brackets, equality sign, and the name of the new value.

For Loops

Iteration is a fundamental building block of all programs. It is the ability to execute a certain code repeatedly.

```
In [ ]: for n in even:
            print n
```



The list “even” contains all the even numbers from 0 to 20. “for n in even”, colon, which would mean *for every* element n in the list “*even*”, do the following: print that element.

The command in the loop body is performed once *for each* element in the *even* list.

While Loops and Incrementing

The same output we obtained in the previous lesson could be achieved after using a while loop, instead of a for loop. However, the structure we will use will be slightly different.

```
In [1]: x = 0
        while x <= 20:
            print x,
            x = x + 2

0 2 4 6 8 10 12 14 16 18 20
```

Initially, we will set a variable x equal to zero. And we'll say: **while** this value is smaller than or equal to 20, print x.
We want to get the loop to end. What is supposed to succeed, the loop body in the "while" block, is a line of code that specifies a change in x or what has to happen to x after it is printed. In our case, we will tell the computer to bind x to a value equal to x + 2.

```
In [2]: x = 0
        while x <= 20:
            print x,
            x += 2

0 2 4 6 8 10 12 14 16 18 20
```

In programming terms, adding the same number on top of an existing variable during a loop is called **incrementing**. The amount being progressively added is called an **increment**. In our case, we have an increment of 2.

The Pythonic syntax offers a special way to indicate incrementing: **x += 2**

Create Lists with the range() Function

When you need to randomize data points and lists with data points, you can use Python's built-in range function.

The syntax of the function is the following:

```
        0          1
range(start, stop, step)

start = the first number in the list

stop = the last value +1

step = the distance between each two consecutive values
```

The stop value is a required input, while the start and step values are optional. If not provided, the start value will be automatically replaced with a 0, and the step value would be assumed to be equal to 1.

Create Lists with the range() Function

```
In [1]: range(10)
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: range(3,7)
Out[2]: [3, 4, 5, 6]

In [3]: range(1,20,2)
Out[3]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

range(10) will provide a list of 10 elements, starting from 0, implied after not indicating a start value, and ending at the tenth consecutive number – 9.

In another cell, if in the "range" function we declare as arguments 3 and 7, for instance, Python will accept 3 as a start value, and 7 as a stop value of the range. So, we'll have 4 elements – 3, 4, 5, and 6.

To specify a step value in a range, the other two arguments must be chosen as well. *range(1,20,2)* creates a list with all the odd numbers from 1 to 19 included. It will start with the number 1, and the list will end with number 19 (which equals the stop value 20 minus 1), stating only the odd numbers.

Use Conditional Statements and Loops Together

```
In [2]: for x in range(20):
            if x % 2 == 0:
                print x,
            else:
                print "Odd",
```

```
0 Odd 2 Odd 4 Odd 6 Odd 8 Odd 10 Odd 12 Odd 14 Odd 16 Odd 18 Odd
```

You create an iteration that includes a conditional in the loop body. You can tell the computer to print all the even values between 0 and 19 and state “Odd” in the places where we have odd numbers.

Let’s translate this into computational steps.

If x leaves a remainder of 0 when divided by 2, which is the same as to say “if x is even”, then print x on the same line. “Else”, which means unless x is even, or if x is odd, print “Odd”.

All In – Conditional Statements, Functions, and Loops

We use iterations when we have to go through variables that are part of a list.

You can count the number of items whose value is less than 20 in a list. First, define a function that takes as an argument numbers, where "numbers" will be a certain list variable. The trick is to create a variable that, so to speak, "departs" from 0. Let's call it total.

```
In [1]: def count(numbers):
            total = 0
            for x in numbers:
                if x < 20:
                    total += 1
            return total
```

The idea is that, when certain conditions are verified, total will change its value. This is why, in such a situation, it is appropriate to call this variable a **rolling sum**.

More technically, when we consider x in the numbers list, if it is smaller than 20, we will increment the total by 1 and finally return the total value. This means that, if x is less than 20, total will grow by 1, and if x is greater than or equal to 20, total will not grow. So, for a given list, this count function will return the amount of numbers smaller than 20.

Appendix: Python 2 vs. Python 3 – the **print** function

"Print" is a function that displays text in the Output field of a cell.

Python 2

```
In [1]: print "Text"
        Text

In [2]: print 35
        35
```

In Python 2.7, "print" is a keyword. If you type *print "Text"* you will see *Text.* If you type *print 35* you will have *35* in the output field.

Python 3

```
In [1]: print ("Text")
        Text

In [2]: print (35)
        35
```

In Python 3, "print" is like most functions. Therefore, the information to be printed must be written between parentheses: *print ("Text")* will produce *Text*, while *print (3)* will display *3* in the output field.

Appendix: Python 2 vs. Python 3 – division

Python 2 and Python 3 divide integers differently.

Python 2

```
In [1]: 16/3
Out[1]: 5
```

In Python 2, the ratio of two integers is always an integer. For example, 16/3 will equal 5.

Python 3

```
In [1]: 16/3
Out[1]: 5.333333333333333
```

Instead, in Python 3, the quotient is converted into a float. So, 16/3 will produce an output of 5.33.

Appendix: Python 2 vs. Python 3 – the range function

In Python 2, the **xrange()** function is similar to **range()**.

Python 2

```
In [1]: range(10)
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: xrange(10)
Out[2]: xrange(10)
```

When used to produce a sequence of values, the outputs of the two built-in functions will differ. More importantly, xrange() turns out to be faster when iterating. Although xrange() could be reasonably applied to the last few lectures in this course, there is no real need to opt for xrange() instead of range() in our lectures. Nevertheless, don't be scared if you see it in more advanced bits of code.

Python 3

```
In [1]: range(10)
Out[1]: range(0, 10)
```

In Python 3, there is only one built-in function, and it is range(). It corresponds to the xrange() function in Python 2.